

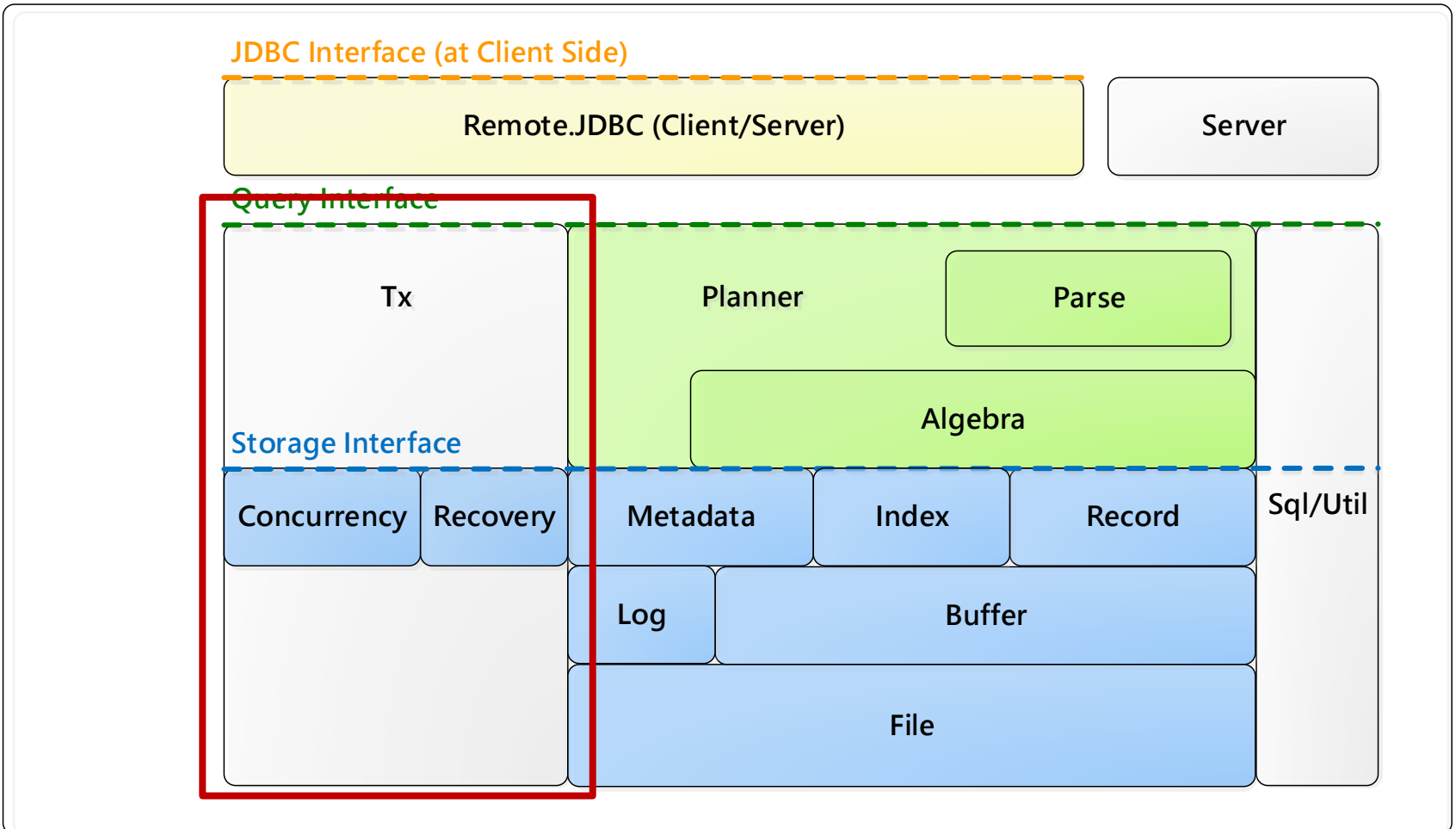


Transaction Management Part I: Concurrency Control

vanilladb.org

Tx Management

VanillaCore



```
VanillaDb.init("studentdb");
```

```
// Step 1
```

```
Transaction tx =  
VanillaDb.txMgr().newTransaction(  
Connection.TRANSACTION_SERIALIZABLE, true);
```

```
// Step 2
```

```
Planner planner = VanillaDb.newPlanner();  
String query = "SELECT s-name, d-name FROM  
departments, "  
+ "students WHERE major-id = d-id";  
Plan plan = planner.createQueryPlan(query,  
tx);  
Scan scan = plan.open();
```

```
// Step 3
```

```
System.out.println("name\tmajor");  
System.out.println("-----\t-----");  
while (scan.next()) {  
String sName = (String) scan.getVal("s-  
name").asJavaVal();  
String dName = (String) scan.getVal("d-  
name").asJavaVal();  
System.out.println(sName + "\t" + dName);  
}  
scan.close();
```

```
// Step 4
```

```
tx.commit();
```

Native API Revisited

- A tx is created upon accepting an JDBC connection
 - by
`VanillaDb.txMgr().newTransaction()`
- Passed as a parameter to Planners/Scanners/RecordFiles



Transaction Manager in VanillaDB

- `VanillaDb.txMgr()` is responsible for creating new transaction and maintaining the active transaction list

TransactionMgr
<u>+ serialConcurMgrCls : Class<?></u> <u>+ rrConcurMgrCls : Class<?></u> <u>+ rcConcurMgrCls : Class<?></u> <u>+ recoveryMgrCls : Class<?></u>
+ TransactionMgr() + onTxCommit(tx : Transaction) + onTxRollback(tx : Transaction) + onTxEndStatement(tx : Transaction) + createCheckpoint(tx : Transaction) + newTransaction(isolationLevel : int, readOnly : boolean) : Transaction + newTransaction(isolationLevel : int, readOnly : boolean, txNum : long) : Transaction + getNextTxNum() : long



Naïve ACID

- C and I: each tx locks the entire DB it access
 - Responsibility of the *Concurrency Manager*
- A and D: Write Ahead Logging (WAL)
 - Responsibility of the *Recovery Manager*

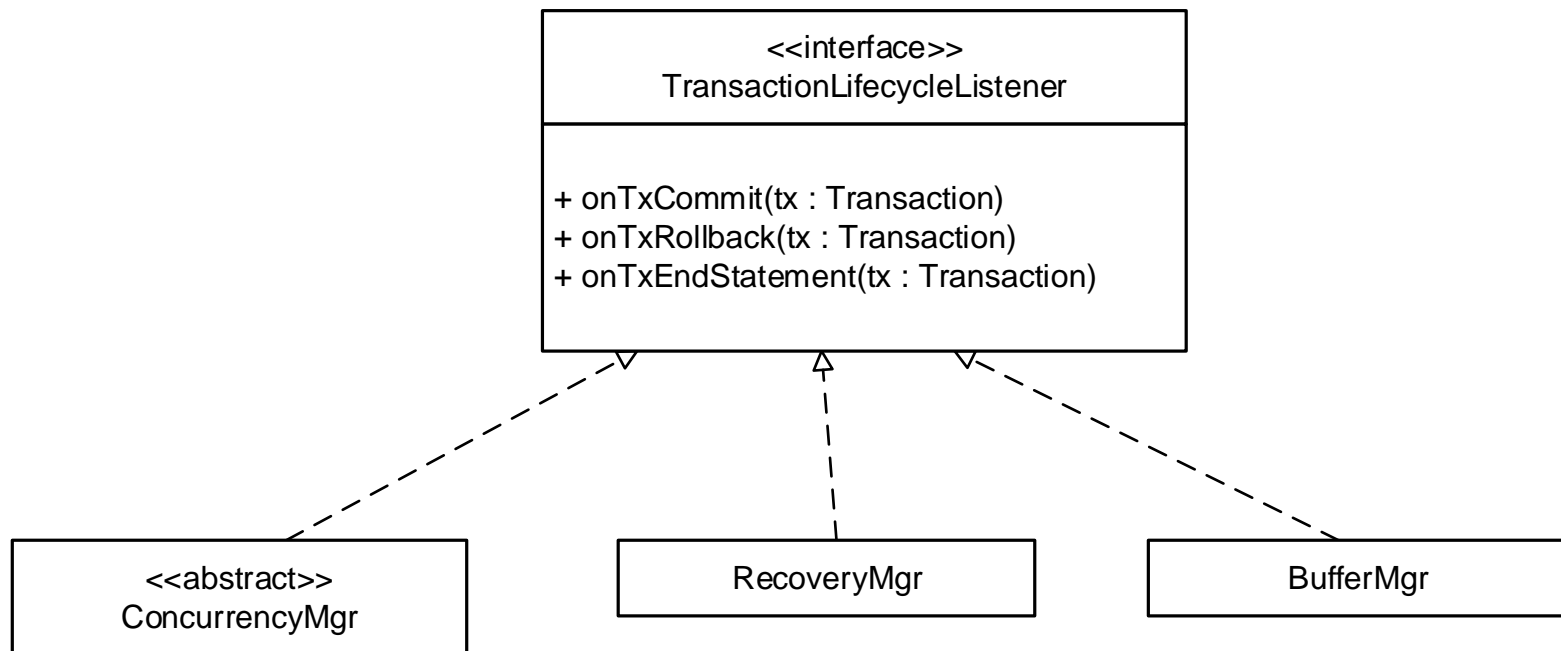
Transaction Management

Transaction
<pre>+ Transaction(concurMgr : ConcurrencyMgr, recoveryMgr : RecoveryMgr, bufferMgr : BufferMgr readOnly : boolean, txNum : long) + addLifecycleListener(l : TransactionLifecycleListener) + commit() + rollback() + endStatement() + getTransactionNumber() : long + isReadOnly() : boolean + concurrencyMgr() : ConcurrencyMgr + recoveryMgr() : RecoveryMgr + bufferMgr() : BufferMgr</pre>

- Each tx has an concurrency and recovery mgr instance
- Transaction life cycle
 - On transaction commit
 - On transaction rollback
 - On transaction end statement

Listeners

- Tx life cycle listener
 - Takes actions to tx life cycle events



Transaction Listener: Concurrency Mgr

- **Serializable** Concurrency manager
 - On tx commit/rollback: releases all locks

```
@Override
public void onTxCommit(Transaction tx) {
    lockTbl.releaseAll(txNum, false);
}

@Override
public void onTxRollback(Transaction tx) {
    lockTbl.releaseAll(txNum, false);
}

@Override
public void onTxEndStatement(Transaction tx) {
    // do nothing
}
```

Transaction Listener: Recovery Mgr

- **Basic** recovery manager
 - On tx rollback: undo all modifications based on log records
 - On tx commit: writes log records to disk and flushes dirty pages

```
@Override
public void onTxCommit(Transaction tx) {
    VanillaDb.bufferMgr().flushAll(txNum);
    long lsn = new CommitRecord(txNum).writeToLog();
    VanillaDb.LogMgr().flush(lsn);
}

@Override
public void onTxRollback(Transaction tx) {
    doRollback(tx);
    VanillaDb.bufferMgr().flushAll(txNum);
    long lsn = new RollbackRecord(txNum).writeToLog();
    VanillaDb.LogMgr().flush(lsn);
}

@Override
public void onTxEndStatement(Transaction tx) {
    // do nothing
}
```

Transaction Listener: Buffer Mgr

- Buffer manager
 - On tx rollback/commit: unpins all pages pinned by the current tx
 - Registered itself as a life cycle listener on start of each tx

```
@Override
public void onTxCommit(Transaction tx) {
    unpinAll(tx);
}

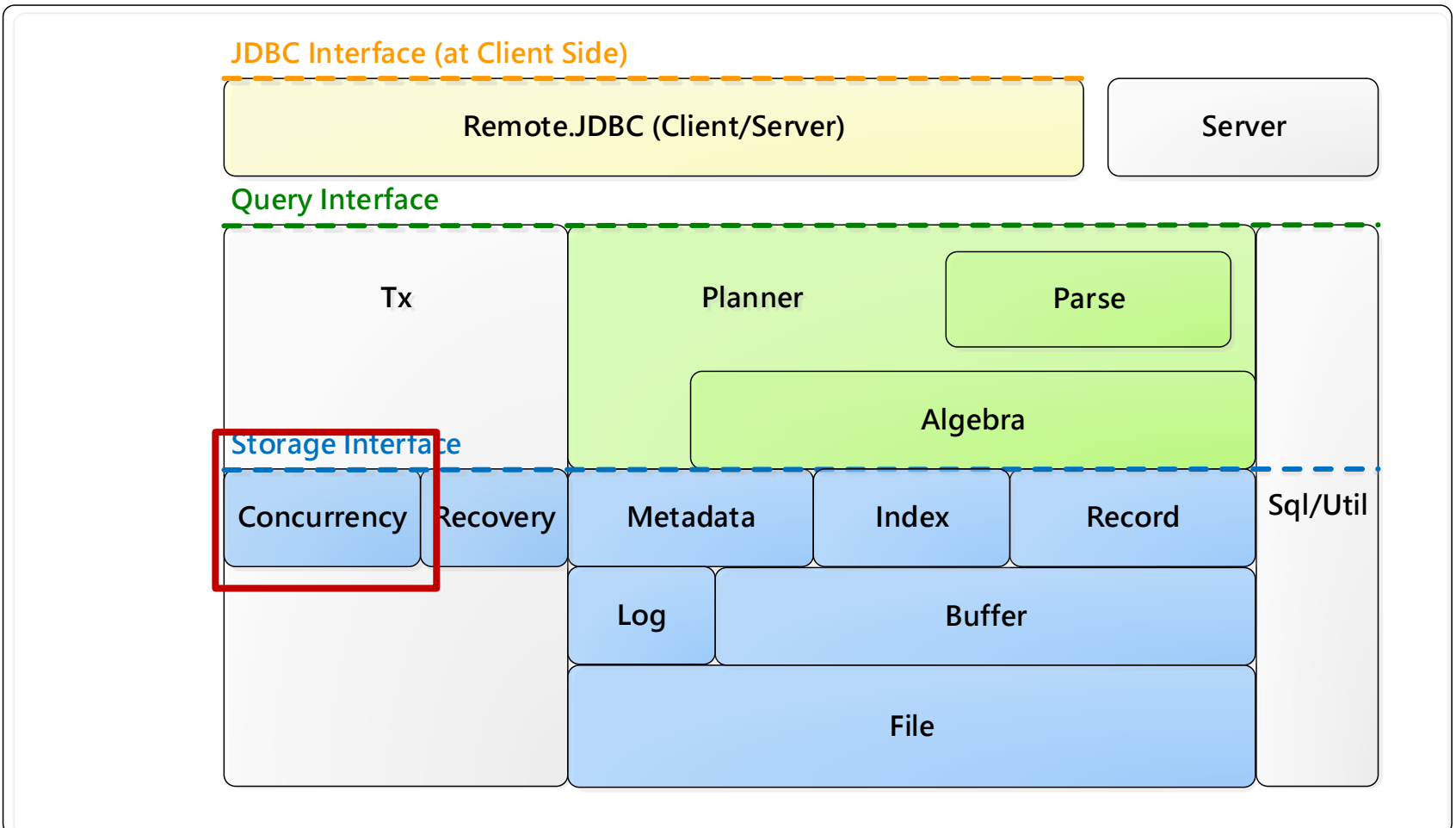
@Override
public void onTxRollback(Transaction tx) {
    unpinAll(tx);
}

@Override
public void onTxEndStatement(Transaction tx) {
    // do nothing
}
```



Today's Focus: Concurrency Mgr

VanillaCore



Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore



Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore



Transaction Management

- The recovery manager ensures atomicity and durability of the database
- How about *consistency* and *isolation*?



Consistency

- ***Consistency***

- Txs will leave the database in a consistent state
- I.e., all integrity constraints are meet
 - Primary and foreign key constrains
 - Non-null constrain
 - (Field) type constrain
 - ...
- Users are responsible for issuing “valid” txs



Isolation

- ***Isolation***

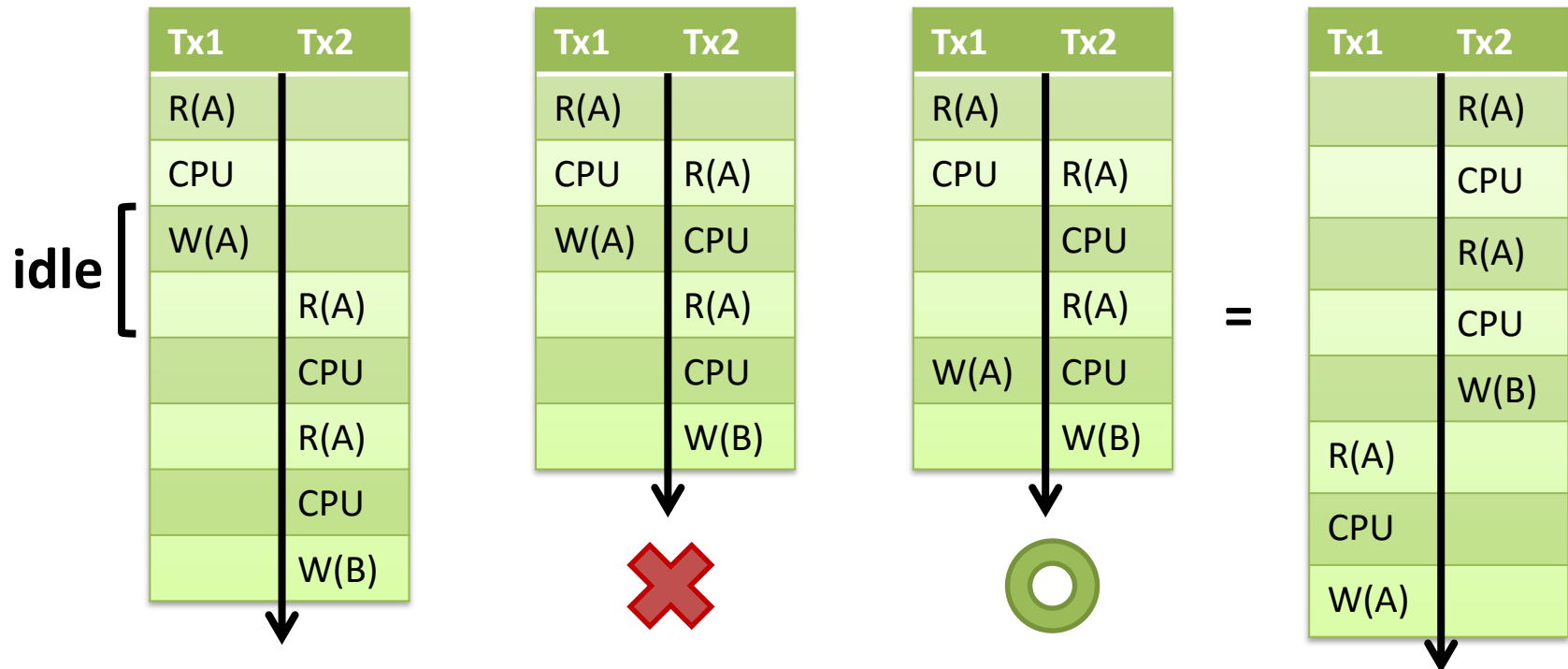
- Interleaved execution of txs should have the net effect identical to executing tx in ***some*** serial order
- T_1 and T_2 are executed concurrently, isolation gives that the net effect to be equivalent to either
 - T_1 followed by T_2 or
 - T_2 followed by T_1
- The DBMS does ***not*** guarantee to result in ***which particular*** order



Why do we need to interleave txs?

Concurrent Txs

- Since I/O is slow, it is better to execute Tx1 and Tx2 concurrently to reduce CPU idle time



- The concurrent result should be the same as serial execution in *some* order
 - Better concurrency



Concurrent Txs

- Pros:
 - Increases throughput
 - Shortens response time for short txs



Transactions and Schedules

- Before executing T_1 and T_2 :
 - $A = 300, B = 400$

T1:	BEGIN	$A=A+100,$	$B=B-100$	END
T2:	BEGIN	$A=1.06*A,$	$B=1.06*B$	END

- Two possible execution results
 - T_1 followed by T_2
 - $A = 400, B = 300 \rightarrow A = 424, B = 318$
 - T_2 followed by T_1
 - $A = 318, B = 424 \rightarrow A = 418, B = 324$



Transactions and Schedules

- A *schedule* is a list of actions/operations from a set of transaction
- If the actions of different transactions are not interleaved, we call this schedule a *serial schedule*

T1: $A = A + 100, \quad B = B - 100$

T2: $A = 1.06 * A, \quad B = 1.06 * B$



Transactions and Schedules

- Equivalent schedules
 - The effect of executing the first schedule is identical to the effect of executing the second schedule
- ***Serializable schedule***
 - A schedule that is equivalent to some serial execution of the transactions



Transactions and Schedules

- A possible interleaving schedule

T1:	$A=A+100,$	$B=B-100$
T2:	$A=1.06*A,$	$B=1.06*B$

– Result: $A = 424, B = 318$

– A serializable schedule

- T_1 followed by T_2

T1:	$A=A+100,$	$B=B-100$
T2:	$A=1.06*A,$	$B=1.06*B$



Transactions and Schedules

- How about this schedule?

T1:	$A = A + 100,$	$B = B - 100$
T2:	$A = 1.06 * A, B = 1.06 * B$	

- Result: $A = 424, B = 324$
 - A non-serializable schedule
 - Violates the isolation
- DBMS's view

T1:	$R(A), W(A),$	$R(B), W(B)$
T2:	$R(A), W(A), R(B), W(B)$	



Outline

- Schedules
- **Anomalies**
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore



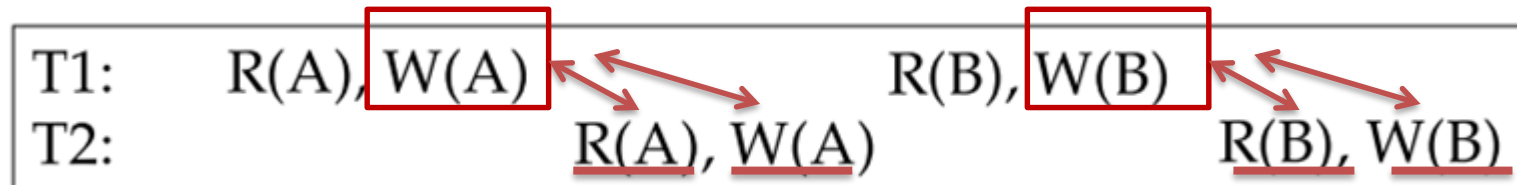
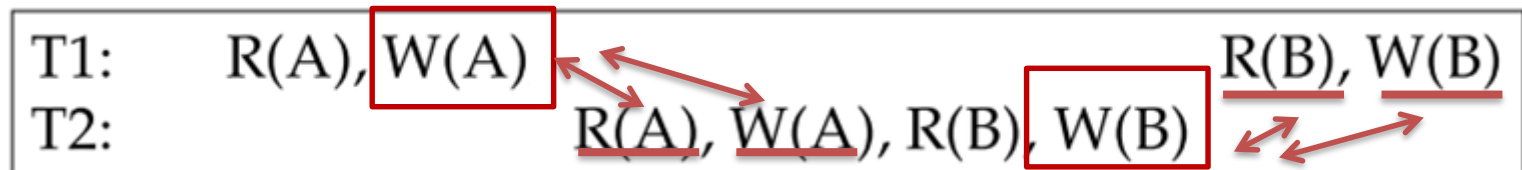
Conflict Operations

- What actions jeopardize the interleaving?
 - Those on the same object and **conflict** with each other
- Conflicts:
 - Write-read conflict (on the same object)
 - Read-write conflict
 - Write-write conflict
- Read-read conflict?
 - No



Conflict Operations

- Two operations on the same object are **conflict** if they are operated by different txs and at least one of these operations is a write



Anomalies due to Write-Read Conflict

- Reading uncommitted data

– *Dirty reads*

T1:	R(A), <u>W(A)</u> ,	R(B), W(B)
T2:	<u>R(A)</u> , W(A), R(B), W(B)	

- A *unrecoverable schedule*

T1:	R(A), W(A),	R(B), W(B), Abort
T2:	R(A), W(A), C	

– T1 cannot abort!

– *Cascading aborts* if T2 completes after T1 aborts

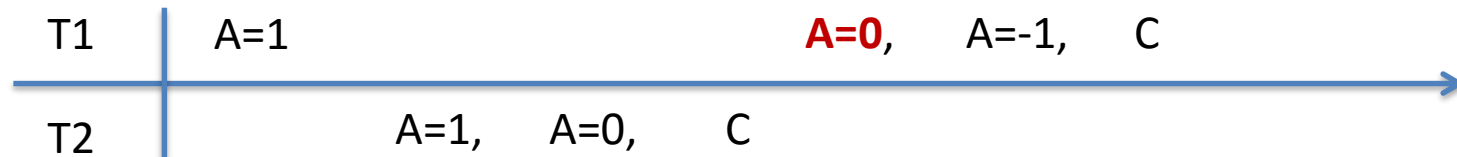


Anomalies due to Read-Write Conflict

- **Unrepeatable reads:**

- T_1 : if $(A > 0) A = A - 1$;
- T_2 : if $(A > 0) A = A - 1$;
- IC on A : cannot be negative

T1:	<u>R(A),</u>	<u>R(A),</u>	W(A), C
T2:		R(A),	<u>W(A),</u> C

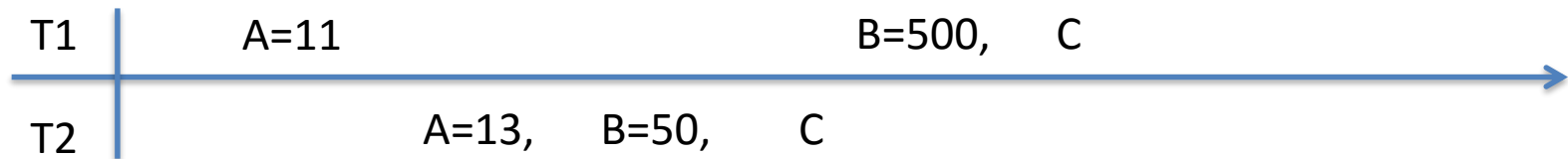


Anomalies due to Write-Write Conflict

- ***Lost updates:***

- $T_1: A = A + 1; B = B * 10;$
- $T_2: A = A + 2; B = B * 5;$
- Start with $A=10, B=10$

T1:	<u>W(A),</u>	W(B), C
T2:	<u>W(A),</u>	W(B), C



Avoiding Anomalies

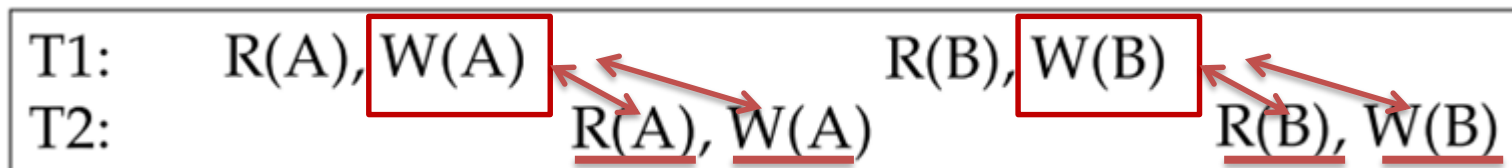
- To ensure serializable schedule:
- How?
 - Perform all conflicting actions between T1 and T2 *in the same order* (either T1's before T2's or T2's before T1's)
 - I.e., to ensure *conflict serializability*



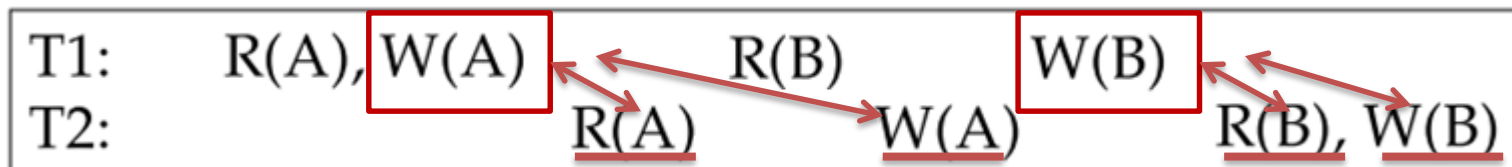
Conflict Equivalent

- If two operations are not conflict, we can **swap** them to generate an equivalent schedule
- Schedule 1 is **conflict equivalent** to schedule 2 and schedule 3

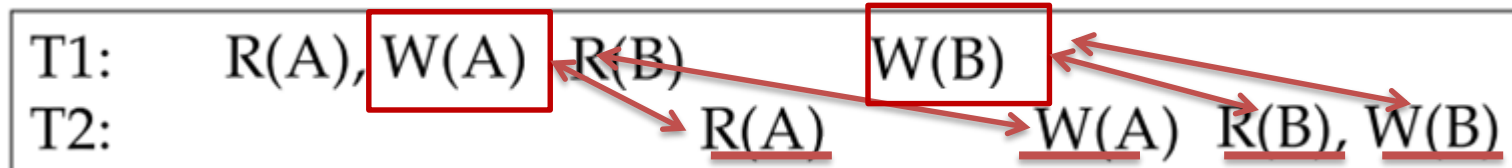
Schedule 1



Schedule 2



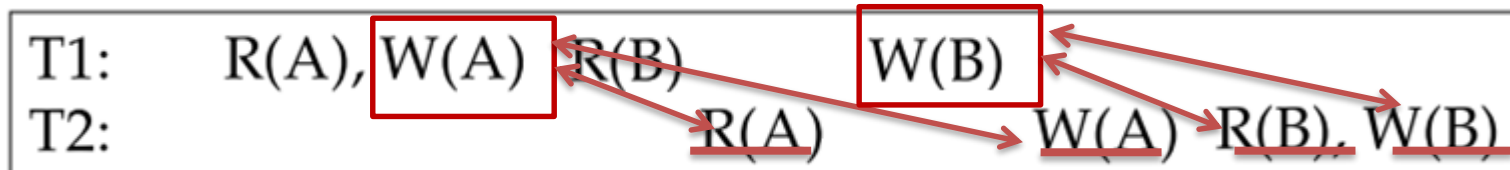
Schedule 3



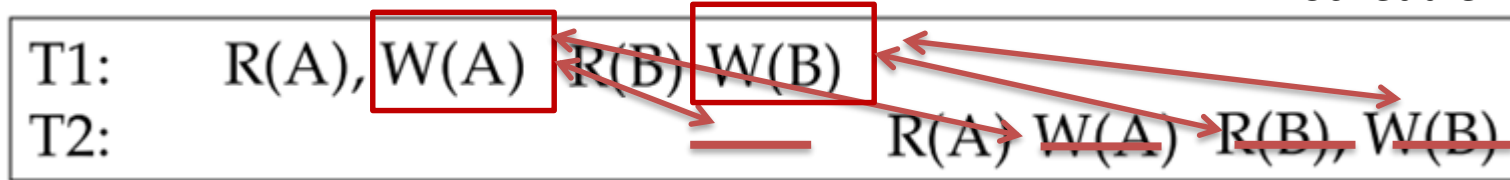
Conflict Serializable

- By swapping non-conflict operations, we can transfer the schedule 1 into a serial schedule 4
- We say that schedule 1 is ***conflict serializable***

Schedule 3



Schedule 4



Ensuring serializability is ***not enough***



Avoiding Anomalies

- We also need to ensure recoverable schedule
- Definition: A schedule is *recoverable* if each tx T commits only after all txs whose changes T reads, commit
- How?
 - Avoid cascading aborts
 - Disallow a tx from reading uncommitted changes from other txs



Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore



Lock-Based Concurrency Control

- For isolation and consistency considerations, the DBMS should only allow *serializable*, *recoverable* schedules
 - Uncommitted changes cannot be seen
 - Cannot overwrite uncommitted change
 - Ensure repeatable read
- A *lock* is a mechanism to control concurrent access to a data item

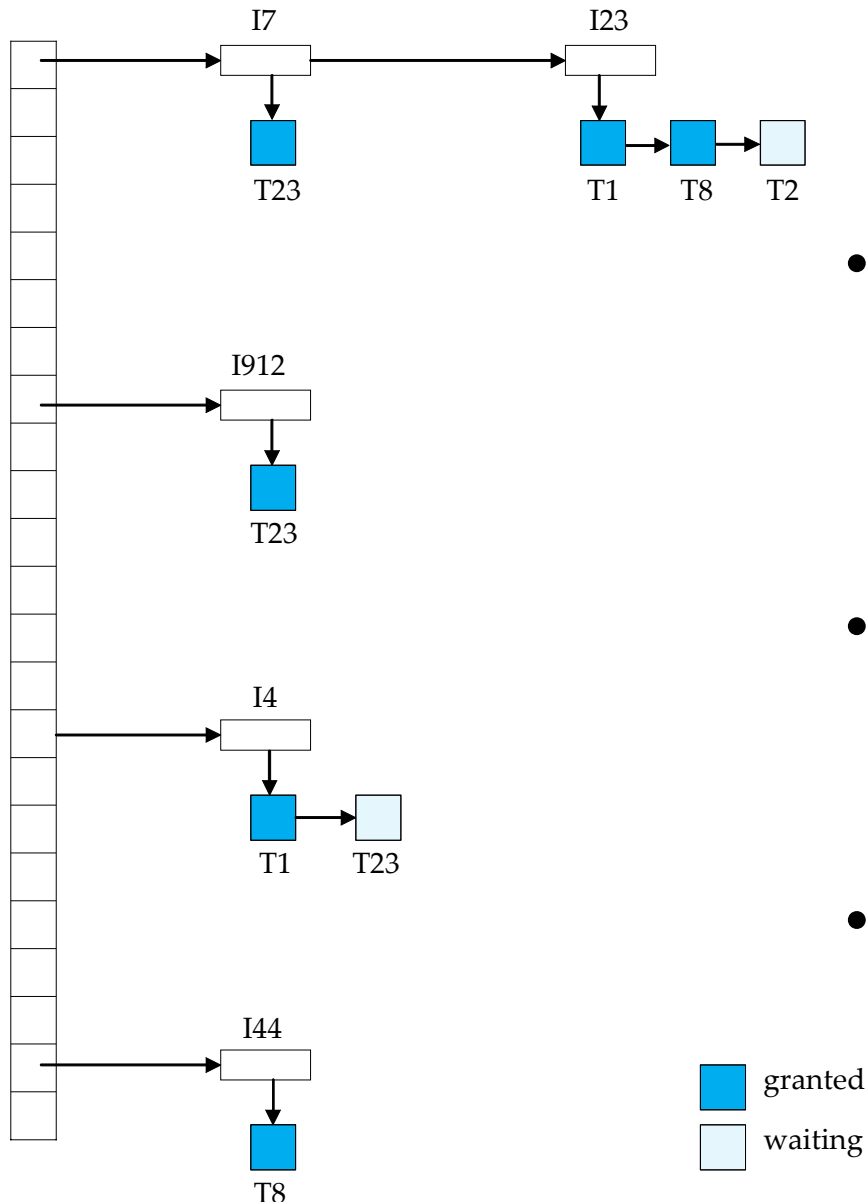


Lock Management

- Lock and unlock requests are handled by the *lock manager*
- Lock table entry
 - Number of transactions currently holding a lock
 - Type of lock held
 - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations



Lock Table



- Implemented as an in-memory hash table indexed on the name of the data item being locked
- New request is added to the end of the queue of requests for the data item
- Lock is granted if it is compatible with all earlier locks



Lock-Based Concurrency Control

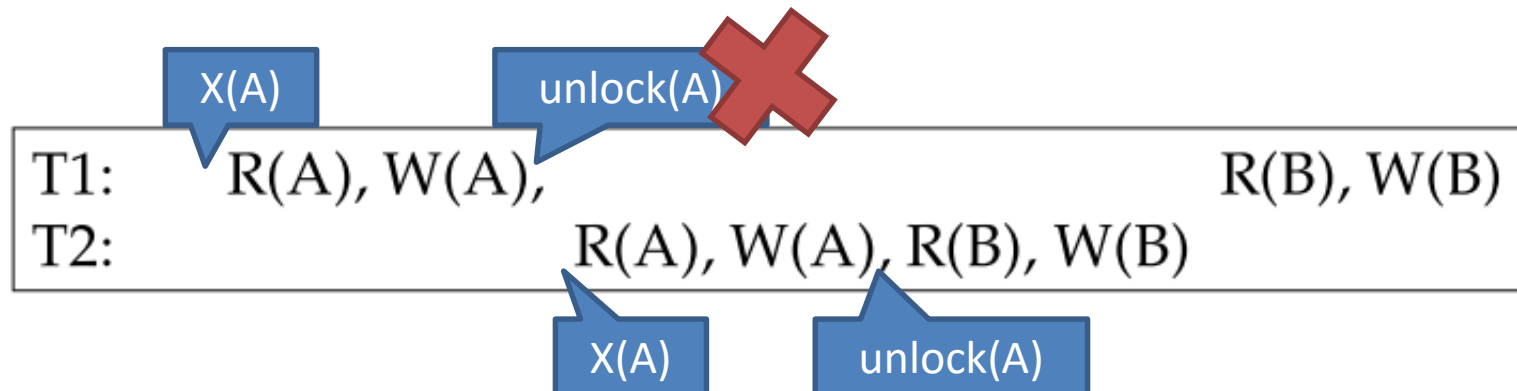
- What kinds of locks do we need?

- *Shared (S) lock*

- *Exclusive (X) lock*

	S	X
S	True	False
X	False	False

- What type of lock to get for each operation?
- When should a transaction acquire/release lock?
- Critical section alike?



Lock \neq latch

- Lock: long-term, tx-level
- Latch: short-term, ds/alg-level

Lock-Based Concurrency Control

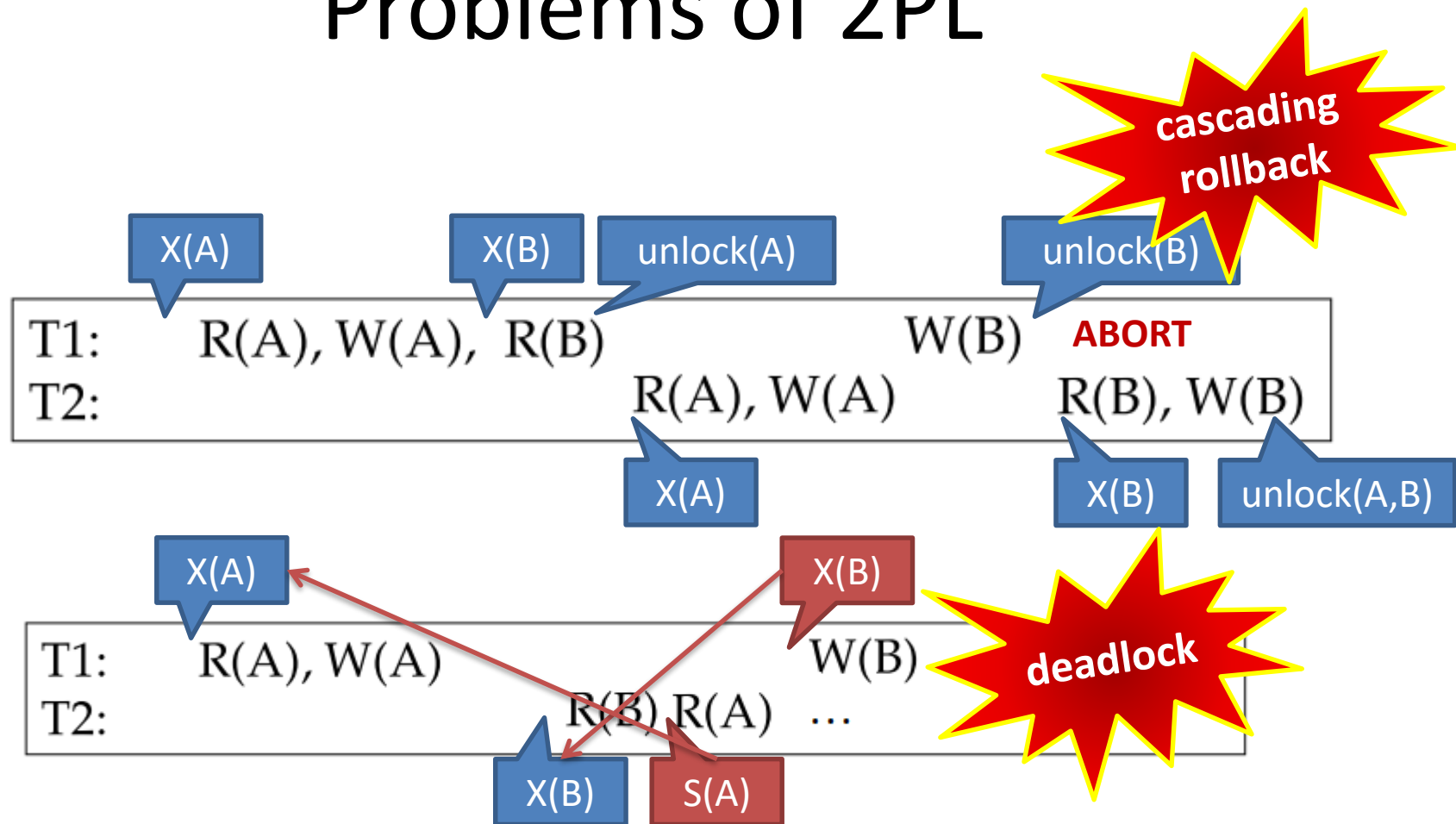
- ***Locking protocol***
 - A set of rules followed by all transactions for requesting and releasing locks
- The ***Two phase Locking Protocol (2PL)***
 - Phase 1: Growing Phase
 - Each tx must obtain an S (X) lock on an object before reading (writing) it
 - Phase 2: Shrinking Phase
 - A transaction can not request additional locks once it releases any locks
- The 2PL can ensure serializable schedules



Is 2PL enough?



Problems of 2PL



- **Starvation** is also possible if concurrency control manager is badly implemented



Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and **S2PL**
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore



Strict Two-Phase Locking

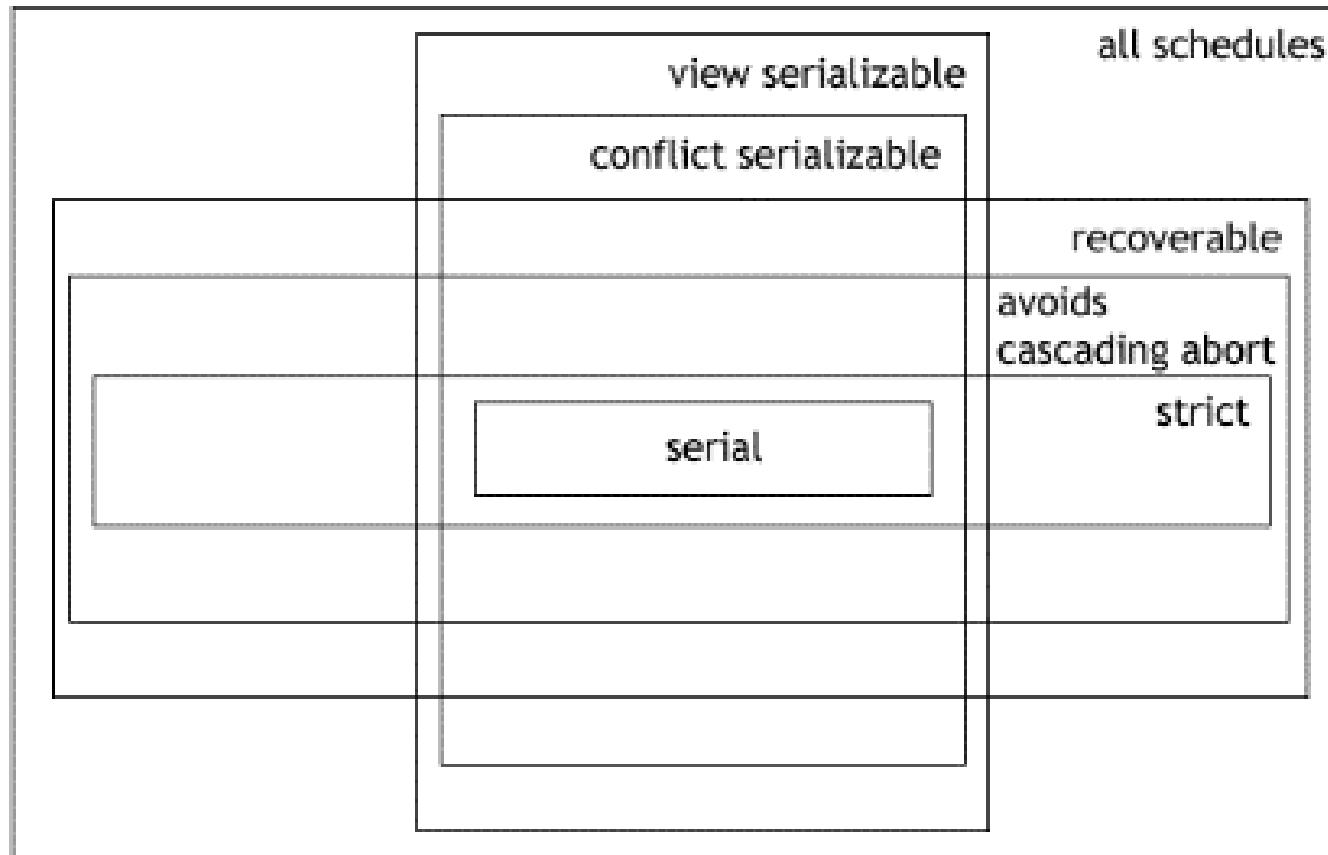
- S2PL
 1. Each tx obtains locks as in the growing phase in 2PL
 2. But the tx *holds all locks until it completes*
- Allows only serializable and *stric* schedules



Strict Two-Phase Locking

- Definition: A schedule is *strict* iff for any two txs T1 and T2, if a write operation of T1 precedes a conflicting operation of T2 (either read or write), then T1 commits before that conflicting operation of T2
 - Strictness \rightarrow no cascading abort (converse not true)
- Avoids cascading rollback, but still has deadlock

Serializability and Recoverability



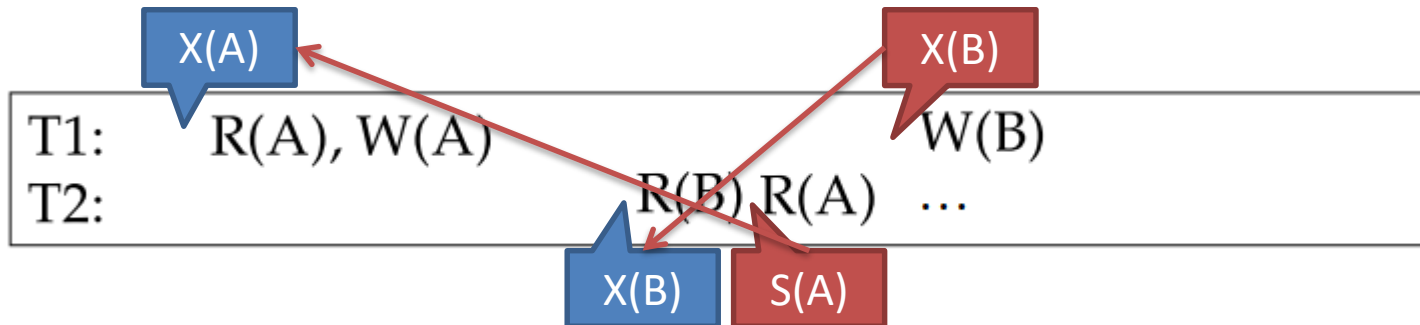
Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - **Deadlock**
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore

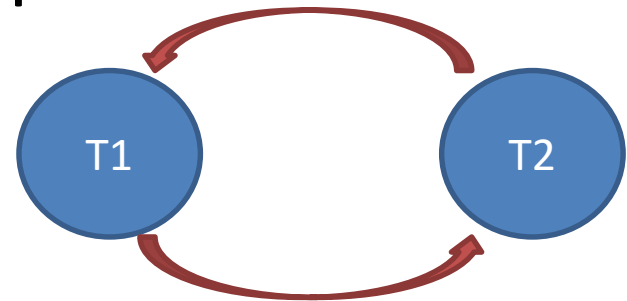


Coping with Deadlocks

- Cycle of transactions waiting for locks to be released by each other



- Detection: ***Waits-for*** graph
 - Detect cycle
- Can be used when acquires locks or buffers



Other Techniques (1)

- **Timeout & rollack** (deadlock detection)
 - Assume T_i wants a lock that T_j holds
 1. T_i waits for the lock
 2. If T_i stays on the wait list too long then: T_i is rolled back
- **Wait-die** (deadlock prevention)
 - Assume each T_i has a priority (e.g., tx number or timestamp)
 - If T_i wants a lock that T_j holds
 1. If T_i has higher priority, T_i waits for T_j ;
 2. otherwise T_i aborts



Other Techniques (2)

- **Conservative locking** (deadlock prevention)
 - Every T_i locks **all objects at once** (atomically) in the beginning
 - No interleaving for conflicting txs---performs well only if there is no/very few long txs (e.g., in-memory DBMS)
 - How to know which objects to lock before tx execution?
 - Requires the coder of a stored procedure to specify its read- and write-sets explicitly
 - Does not support ad-hoc queries



You Have Assignment!

Assignment: Conservative Locking

- Implement a ConcurrencyMgr running the conservative locking protocol
 - Modify the stored procedure API to accommodate read-/write-sets



Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore



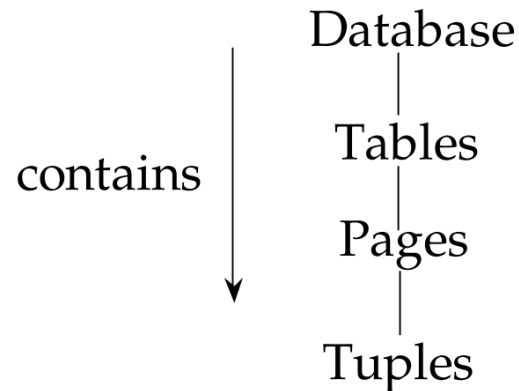
Granularity of Locks

- What “objects” to lock?
 - Records vs. blocks vs. tables/files
- **Granularity** of locking objects
 - Fine granularity: high concurrency, high locking overhead
 - Coarse granularity: low locking overhead, low concurrency



Reducing Locking Overhead

- Data “containers” are nested



- When scanning, can we lock a file instead of all contained blocks/records to reduce the locking overhead?

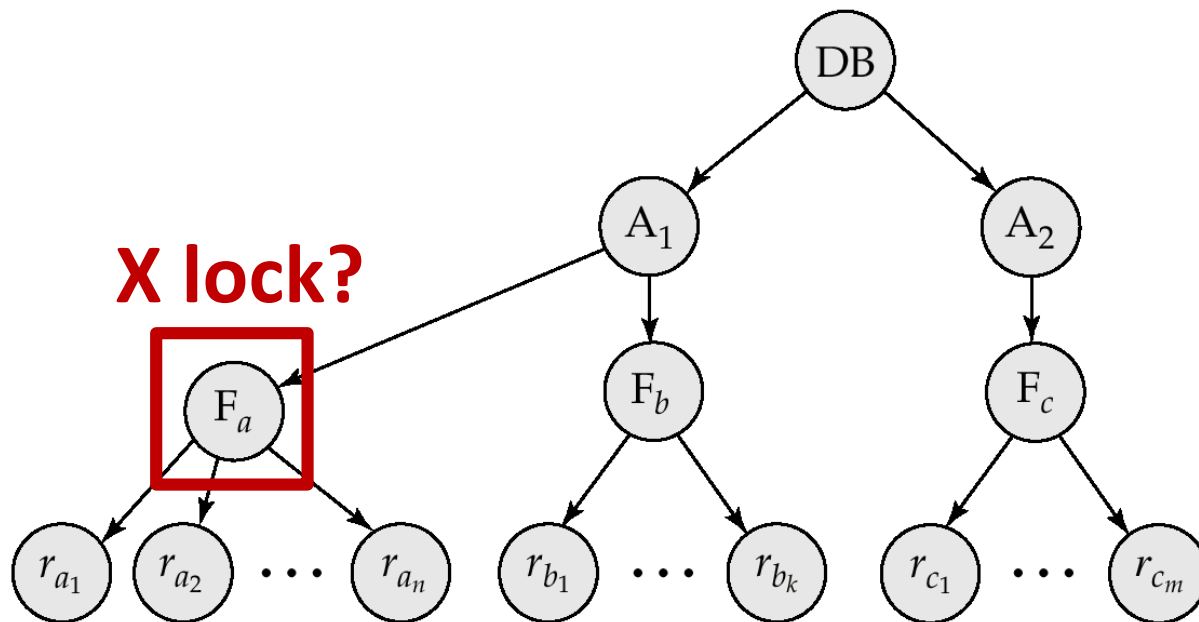
Multiple-Granularity Locks

- ***Multiple-granularity locking*** (MGL) allows users to set locks on objects that contain other objects
 - Locking a file implies locking ***all*** contained blocks/records
- How can the lock manager know whether a file is lockable?
 - Some other tx may hold a conflicting lock on a block in that file



Checking If An Object Is Locked

- To lock a file, check whether all blocks/records in that file are locked
 - Good strategy?
- Does **not** save the locking overhead



Multiple-Granularity Locks

- Allow transactions to lock at each level, but with a special protocol using new “intention” locks:
- **Intention-shared** (IS)
 - Indicates explicit locking at a lower level of the tree but only with shared locks
- **Intention-exclusive** (IX)
 - Indicates explicit locking at a lower level with exclusive or shared locks
- **Shared and intention-exclusive** (SIX)
 - The subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks



Multiple-Granularity Locks

- The compatibility matrix for all lock modes is:

	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
S IX	✓	×	×	×	×
X	×	×	×	×	×



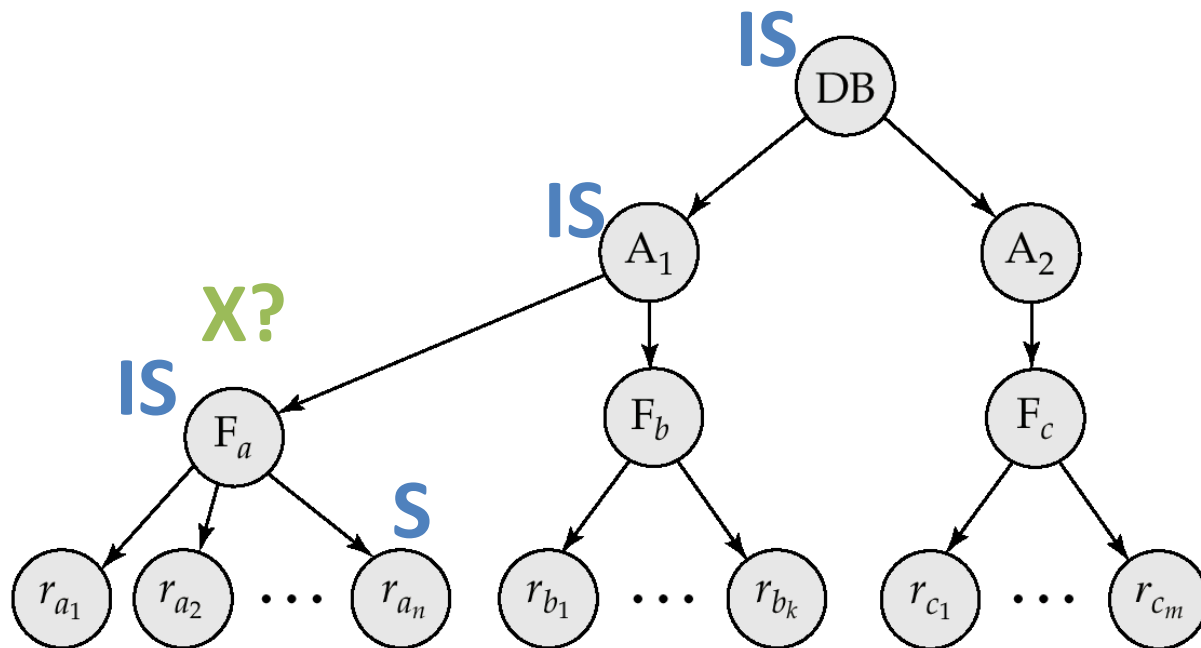
Multiple Granularity Locking Scheme

- Transaction T_i can lock a node Q , using the following rules:
 1. The lock compatibility matrix must be observed
 2. The root of the tree must be locked first, and may be locked in any mode
 3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode
 4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode
 5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
 6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i



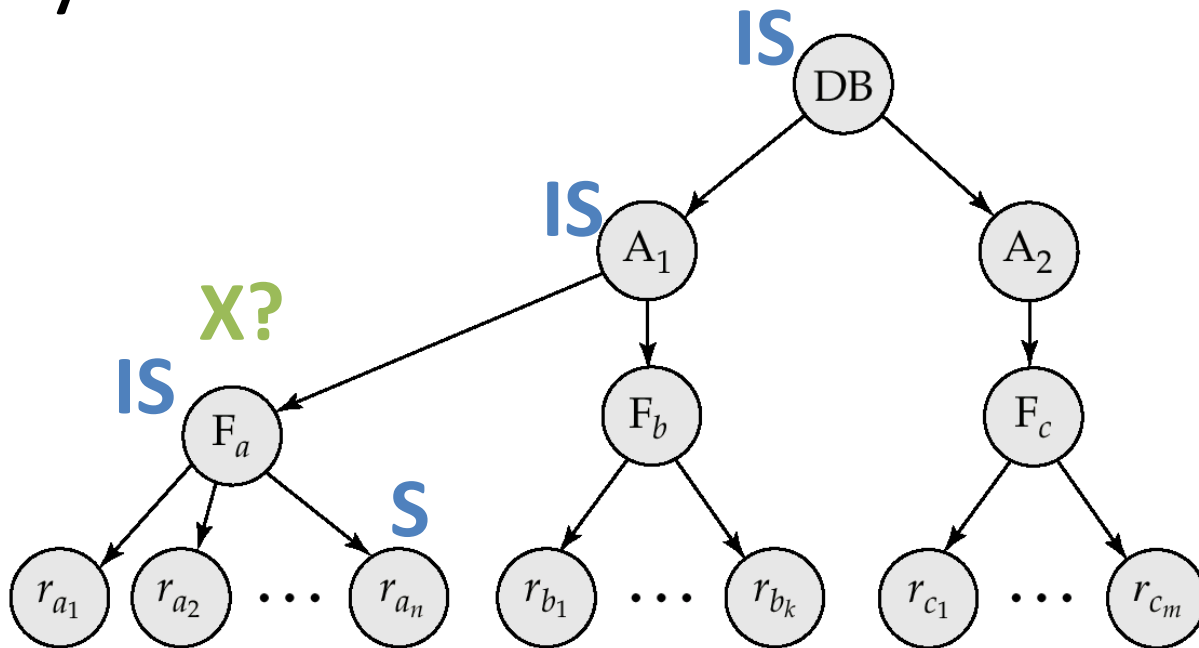
Acquiring Locks in MGL: An Example

- Locks are acquired in *root-to-leaf* order
 - Tx1 wants to share-lock a record
 - Tx2 wants to exclusive-lock a file



Releasing Locks in MGL

- Locks need to be released in *leaf-to-root* order
- Why?



Usage Examples of MGL

- T_1 scans R, and updates a few tuples:
 - T_1 gets an SIX lock on R, and occasionally get X lock on the tuples under modification
- T_2 uses an index to read only part of R:
 - T_2 gets an IS lock on R, and repeatedly gets an S lock on a tuple of R
- T_3 reads the size of R:
 - T_3 gets an S lock on R



Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- **Dynamic databases**
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore



Dynamic Databases

- So far, we have treated a database as a fixed collection of independent data objects
 - Only reads and writes
- However, the database can grow and shrink through the insertions and deletions
- What trouble will the insertions and deletions make?
 - *Phantoms*



Phantoms Caused by Insertion

- T_1 : `SELECT * FROM users WHERE age=10;`
- T_2 : `INSERT INTO users
VALUES (3, 'Bob', 10); COMMIT;`
- T_1 : `SELECT * FROM users WHERE age=10;`
- A transaction that reads the entire contents of a table multiple times will see different data
 - E.g., in a join query



Phantoms Caused by Update

- T_1 : `SELECT * FROM users WHERE age=10;`
- T_2 : `UPDATE users SET age=10 WHERE id=7;`
`COMMIT;`
- T_1 : `SELECT * FROM users WHERE age=10;`
- T_1 only share locks the records with the age equals to 10
- The record with `id=7` is not in the locking item set of T_1 , so T_2 can update this record



How to Prevent Phantoms?

- EOF locks or multi-granularity locks
 - X-lock the containing file when inserting/updating records in a block
 - Hurt performance (due to serial inserts/updates)
 - Usually used to prevent phantoms by insert
 - But **not** phantoms by update (why?)
- Index (or predicate) locking
 - Prevent phantoms by both causes
 - If indices for the inserting/updating fields are created



Phantom and Conservative Locking

- How about your assignment?
- You should at least implement MGL to prevent phantom due to inserts



Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore



Transaction Characteristics

- SQL allows users to specify the followings:
- ***Access model***
 - READ ONLY or READ WRITE
 - By `Connection.setReadOnly()` in JDBC
- ***Isolation level***
 - Trade anomalies for better tx concurrency
 - By
`Connection.setTransactionIsolation()`



Isolation Levels

- Defined by the ANSI/ISO SQL standard

Isolation level	Dirty reads	Unrepeatable reads	Phantoms
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Read	No	No	Maybe
Serializable	No	No	No

- How to relate this with locking?

Isolation Levels

- Defined by the ANSI/ISO SQL standard

Isolation level	Dirty reads	Unrepeatable reads	Phantoms
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Read	No	No	Maybe
Serializable	No	No	No

Isolation level	Shared Lock	Predicate Lock
Read Uncommitted	No	No
Read Committed	Released early	No
Repeatable Read	Held to completion	No
Serializable	Held to completion	Held to completion



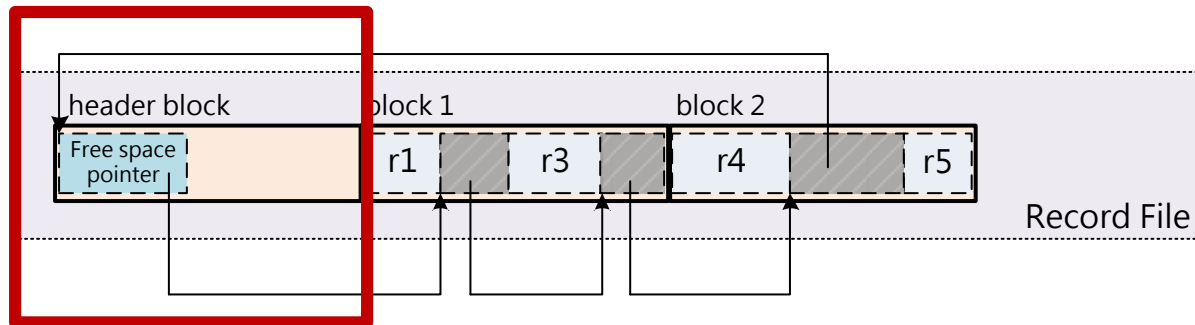
Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- **Meta-structures**
- Concurrency manager in VanillaCore



Meta-Structures

- DBMS maintains some meta-structures in addition to data perceived by users
 - E.g., `FileHeaderPage` in `RecordFile`



- Indices (to be discussed later)

Concurrency Control of Access to Meta-Structures

- Access to `FileHeaderPage`?
 - Whenever insertions/deletions of records happen
- How to lock `FileHeaderPage`?
 - S2PL?
- S2PL will serialize all insertions and deletions
 - Hurts performance if we have many inserts/deletes
 - Spoils index/predicate locking (if any)



Early Lock Release

- Actually, lock of `FileHeaderPage` can be *released early*
- Locking steps for a (logical) insertion/deletion:
 - Acquire locks of `FileHeaderPage` and target object (`RecordPage` or a record) in order
 - Perform changes
 - *Release* the lock of `FileHeaderPage` (but not the object)
- Better I
- No harm to C
- Needs special care to ensure A and D



Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore



Concurrency Manager

- In `storage.tx.concurrency`
- Lock-based protocol
 - MGL granularities: file, block, and record
 - S2PL
 - Deadlock detection: time-limit
- Support txs at different isolation levels
concurrently
 - Serializable
 - Repeatable Read
 - Read Committed



Lock Mode and Access Path

- Access path of a full table scan: a sequence of data block

Prevent phantoms due to inserts, but not updates

	Read rec	Modify/delete rec	Insert rec
SERIALIZABLE	IS lock on file IS lock on block S lock on record	IX lock on file IX lock on block X lock on record	X lock on file X lock on block X lock on record
REPEATABLE READ	IS lock on file and block, release IS on file and block immediately S lock on record	IX lock on file IX lock on block X lock on record <i>Read committed and avoid cascading abort</i>	X lock on file X lock on block X lock on record



Lock Mode and Access Path

	Read rec	Modify/delete rec	Insert rec
READ COMMITTED	IS lock on file and block, release IS on file and block immediately S lock on record and release it upon end statement	IX lock on file IX lock on block X lock on record <i>Early lock release</i>	X lock on file X lock on block X lock on record



Concurrency Manager

- Decide what locks to obtain along the access path

<code><<abstract>> ConcurrencyMgr</code>
<code># txnum : long # locktbl : Locktable</code>
<code><<abstract>> + modifyFile(fileName : String) <<abstract>> + readFile(fileName : String) <<abstract>> + insertBlock(blk : BlockId) <<abstract>> + readBlock(blk : BlockId) <<abstract>> + modifyBlock(blk : BlockId) // methods for B-tree index locking ...</code>



Concurrency Manager

- CCMgr for three isolation levels
 - `SerializableConcurrencyMgr`
 - `RepeatableRead1ConcurrencyMgr`
 - `ReadCommittedConcurrencyMgr`
- Every transaction has its own concurrency managers corresponding to the isolation level



Lock Table

- Implements the compatibility table
- Use time-limit strategy to resolve deadlock

LockTable
<u><<final>> ~ IS LOCK : int</u> <u><<final>> ~ IX LOCK : int</u> <u><<final>> ~ S LOCK : int</u> <u><<final>> ~ SIX LOCK : int</u> <u><<final>> ~ X LOCK : int</u>
<<synchronized>> ~ sLock(obj: Object, txNum : long) <<synchronized>> ~ xLock(obj: Object, txNum : long) <<synchronized>> ~ sixLock(obj: Object, txNum : long) <<synchronized>> ~ isLock(obj: Object, txNum : long) <<synchronized>> ~ ixLock(obj: Object, txNum : long) <<synchronized>> ~ release(obj: Object, txNum : long, lockType : int) <<synchronized>> ~ releaseAll(txNum : long, sLockOnly : boolean)



References

- Database Design and Implementation, chapter 14. Edward Sciore.
- Database management System 3/e, chapter 16. Ramakrishnan Gehrke.
- Database system concepts 6/e, chapter 15, 16. Silberschatz.
- Derby Developer's Guide: Locking, concurrency, and isolation.
 - <http://db.apache.org/derby/docs/10.9/devguide/cdevconcepts30291.html>
- IBM DB2 document: Locks and concurrency control
 - <http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.perf.doc%2Fdoc%2Fc0005266.html>

